# Problem Classes in Intelligent Network Database Design

Juha Taina

University of Helsinki, Department of Computer Science
P.O. Box 26 (Teollisuuskatu 23)
FIN-00014 University of Helsinki, Finland
Phone: *int* + 358 0 708 4247
Fax: *int* + 358 0 708 4441
Email: taina@cs.helsinki.fi

July 26, 1994

**Abstract**

The demands to an IN database make it a distributed real-time heterogeneous database, where each of the nodes can be a parallel database. We will discuss the problems that the IN structure and its demands create, and how the current database research can fulfill them. Our interests are in general solution level, logical data model, query languages, speed, and transaction handling and recovery.

## 1 Preface

[ElNa89] In time of faster service creation and high competition, it is necessary to have open and compatible network systems [AiPW91]. The telephone network solution to the dilemma is both Intelligent Network (IN), that creates the background for new services, and Telecommunication Management Network (TMN), that creates the general telephone network model. Together they form the needed environment [ApKS93]. Our interests are in IN, and especially the database that supports the information exchange in it.

In IN, the database must be able to handle 95 % of the requests in less than 15 ms, and the total unreachable time in a year must not exceed 10 seconds. If these demands are not met, customers will notice delays in the telephone interface. In this article we will examine the demands, and what solutions we think the current database research can offer.

The rest of the paper is divided into seven chapters. Chapter 2 is a short introduction to IN. Chapter 3 deals the database theory in general level. Chapter 4 is about possible logical data models of an IN database. Chapter 5 is about query languages, both on maintenance and on user level. Chapter 6 is about speed demands and how they can be met. Chapter 7 is about transaction processing and recovery from failure conditions. Chapter 8 is summary.

## 2 Introduction to IN

An Intelligent Network (IN) is able to separate the specification, creation, and control of telephony services from physical switching networks [HoHa92]. IN and its standardization is also the first step to create an open standardized telephone network where new services can be introduced in more rapid speed and systems are compatible with each others. In the future the trend will be more to open systems, where calls don't have to come from the same network ([Hade90],[LoKM90]). This cannot be handled without clear standardization and well defined protocols.

An Intelligent Network consist of the following parts (figure 1): Service Switching Points (SSP), Intelligent Peripherals (IP), Service Control Points (SCP), and Service Data Points (SDP). A SSP accepts a phone call. It requests the caller profile from the SDP database, and then waits for the caller to dial a number. If the number needs IN abilities it triggers a SCP. A SCP executes service creation blocks that are needed for the triggered services. It might or might not consult one or more SDPs for getting the information. The results are returned to the SSP. It is also possible that the SSP triggers an IP, that is responsible of tasks like customized and concatenated voice announcements, voice recognition, and Dual Tone Multi-Frequencies digit collection [MoMa94].

Our focus is in SDPs and how they interact with the other IN objects. A SDP is responsible of keeping and maintaining data that IN needs. Together, all SDPs form a complete distributed IN database.

The requirements that the distributed database must fulfill are not easy. The call profile can vary a lot, and yet it should look transparent to the customer. Let's suppose that Mr. Virtanen wants to make a phone call. He picks the telephone, dials the numbers and waits for the dial tone. He expects the procedure to happen without noticeable delays. If the dial tone is immediate he thinks that he made an error somewhere. If the dial tone is delayed for more than a second or two he gets frustrated to waiting. However, the call might trigger very different operations in the telephone network. In the simplest case the call doesn't need IN abilities, or doesn't need to make queries to a service data point (SDP). In a more complex case the call needs to make one or more SDP queries that might trigger more queries to other SDPs (figure 2). In the worst case it is possible that the information have to be queried from several database nodes. In all cases the response time should be within reasonable limits.

## 3 High-level database model

A possible database solution can be distributed to several nodes on two levels: visible and node level. In the visible level a number of possibly heterogeneous databases work together to create the whole distributed IN database. In the node level each visible node consists of several distinct subnodes. The division between the two levels helps to keep the whole database reliable and to reduce recovery times. With only one level, either the information in the databases would be unreachable during a node recovery, or the information in every node would be replicated to the others.

A typical undistributed database is already quite a complex system (figure 3. It has several interfaces to different types of users that work on different level, several processes, and at least two data stores.

The interfaces vary from system maintenance command interface to a simple query interface. Typically not all users from all interfaces have access to all data. In an IN database the most important interfaces are the maintenance interfaces and application program interfaces. The latter is especially important, as it is the main interface for IN queries.

The database processes handle query processing and database access. The system is

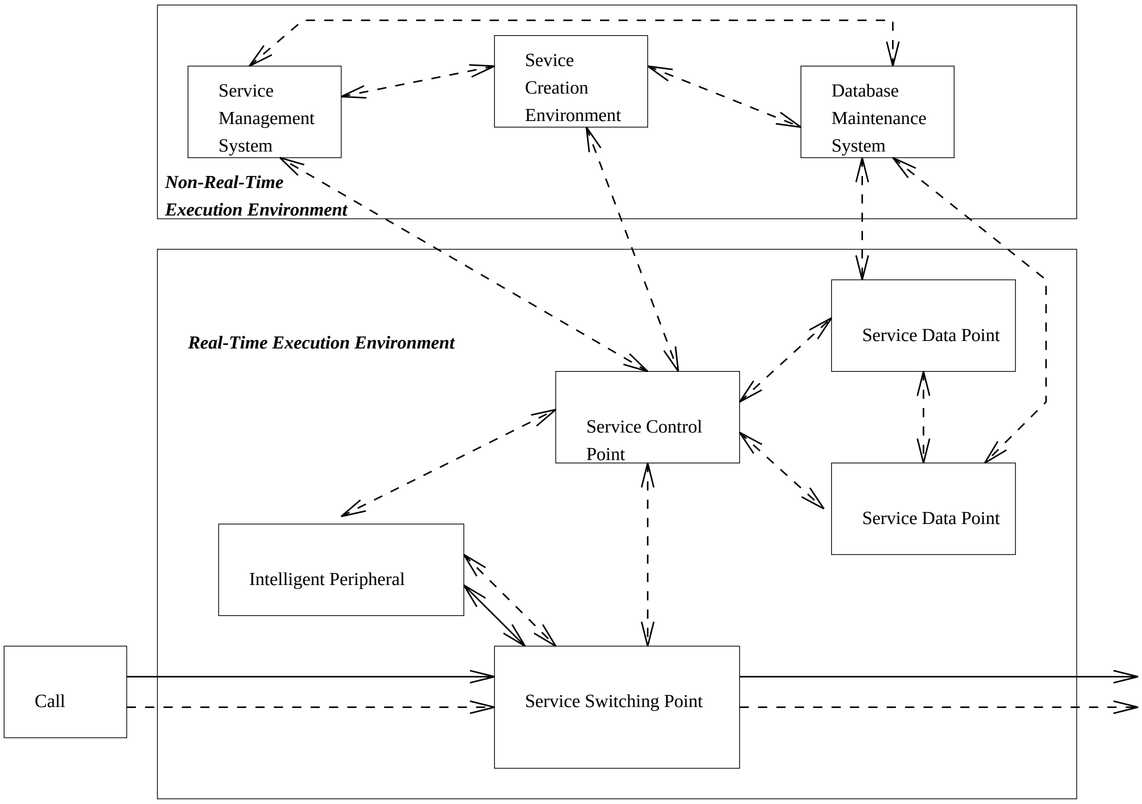


Figure 1: Intelligent Network overview. Dotted lines show signaling, solid lines show transport (original picture without SDP-objects is in [HoHa92])

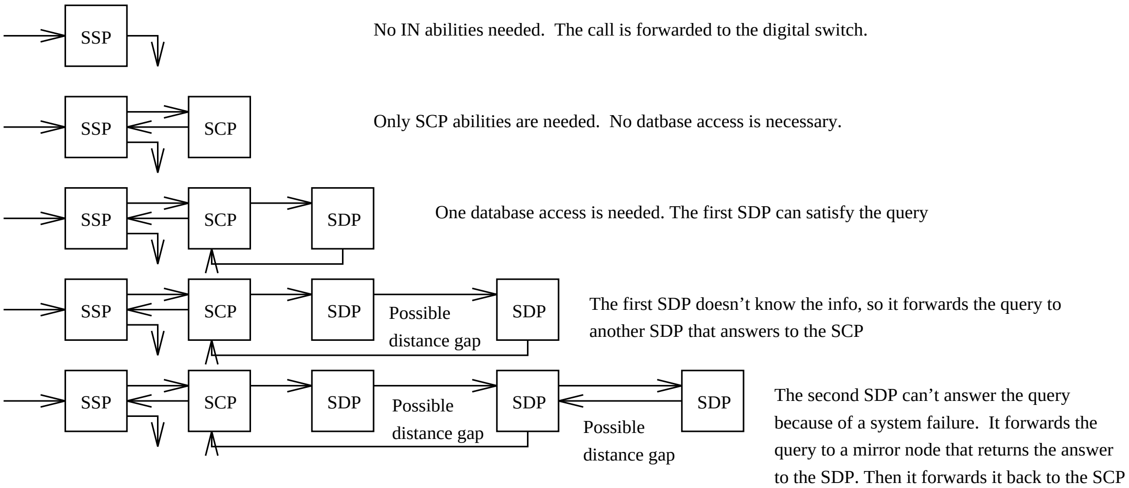


Figure 2: Different examples of call and signal flows in different calls

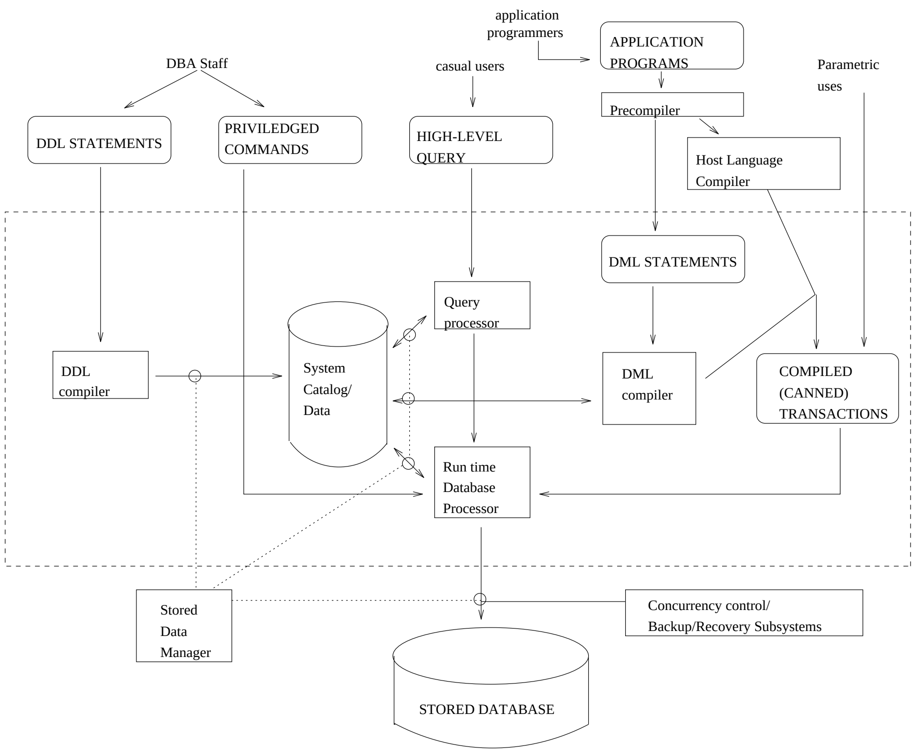


Figure 3: Components of a DBMS. Dotted lines show access under the control of the stored data manager [ElNa89].

hierarchical the way that the lowest level handles direct access to physical data, while the higher level processes use the services that the lower level offers. The most important process is the run time database processor that handles processed queries and updates. Another important process is the DDL compiler that translates system maintenance commands to the database language.

The two necessary data stores are the physical data store and the system catalog. The former has the data itself, the latter has the interpretation of data. The separation between physical data and its interpretation is important as it allows different data to be stored in uniform way. Only the system catalog contents have to be changed. Both the system catalog and the stored database are controlled by the stored data manager, which can also control disks.

The stored data manager is not necessarily a database process. It can handle requests from other programs as well. Other low level subsystems that are outside the database management processes are the concurrency control and backup/recovery subsystems. The former is responsible of keeping simultaneous users unaware of each other. The latter is responsible of keeping stored data available after a system crash. Sometimes the recovery subsystem is also responsible of keeping as much of the database available during the recovery.

The picture becomes more complex when several database nodes interact together. Such systems are called distributed databases. The idea behind a distributed database is simple. Every node is a complete database itself, and they communicate with each other. Thus the figure 3 is valid to a single node, although there is a new component that takes care of the distribution algorithms. Alternatively it is possible to divide the tasks into different components, and only add a low level component that takes care of communication to other nodes.

In IN the database architecture is basically a distributed real-time database where each node is a parallel database. The database nodes don't have to be homogeneous. Thus the total picture is a real-time distributed heterogeneous database where nodes can be parallel databases.

There are at least three different levels we can design and analyze the system. First there is the total database level, where the group of SDPs creates the total database. Then there is the node level, where each of the SDPs creates a database. In such a level the databases are usually parallel databases with enough replication. And finally, there is the parallel database node level, where each of the parallel nodes is a regular database, like in figure 3.

In a parallel database data and processing are divided into several processors that interact with each other. The parallel nature of a database node creates new processes or alters the current ones. There are three different kind of parallel databases, depending on the level of resource sharing: shared nothing systems, shared disk systems and shared all systems (figure 4).

In a shared nothing system each of the processors have private memory and disk. The whole system consists of distinct computers that communicate over a fast network. The connected computers are called parallel database nodes.

Usually one of the nodes is a leader that shares the tasks to the other nodes. Not all the nodes need to be connected to each other, but each of them can be reached from the others.

The advantage of a shared nothing system is that the parallel computing makes complex queries easier to handle, as long as the problems are divisible to smaller subtasks [Seli90]. It is also a working solution if queries can be scheduled to different nodes without overloading any of them. A problem is that queries don't usually access data evenly, and thus some of the nodes are overworked. Also sometimes parallel power is lost due to the overhead that is needed to node communication and task sharing.

In a shared disk system the nodes have private memories, but disks are shared. The communication between different processes is simpler than in a shared nothing system because disks can be used to message sharing. However the shared disks themselves can become a bottleneck.

In a shared all system both the main memory and the disks are shared. The communication channels can be implemented to the main memory, but then both the disks and the memory can become a bottleneck.

An IN database node fills the requirements of a shared nothing database. The IN queries are relatively short and in principle it should be possible to cluster data between subnodes the way that queries can be distributed relatively evenly. The number of nodes in a parallel database can grow higher than in shared disk and shared nothing systems. Also the parallel nodes can be regular computers, which makes the implementation cheaper and better suited to an open environment.

# 4 Logical data model

The logical data model must be flexible enough to handle the demands of the network protocols and the information models. Information modeling will be object oriented [CCIT92],

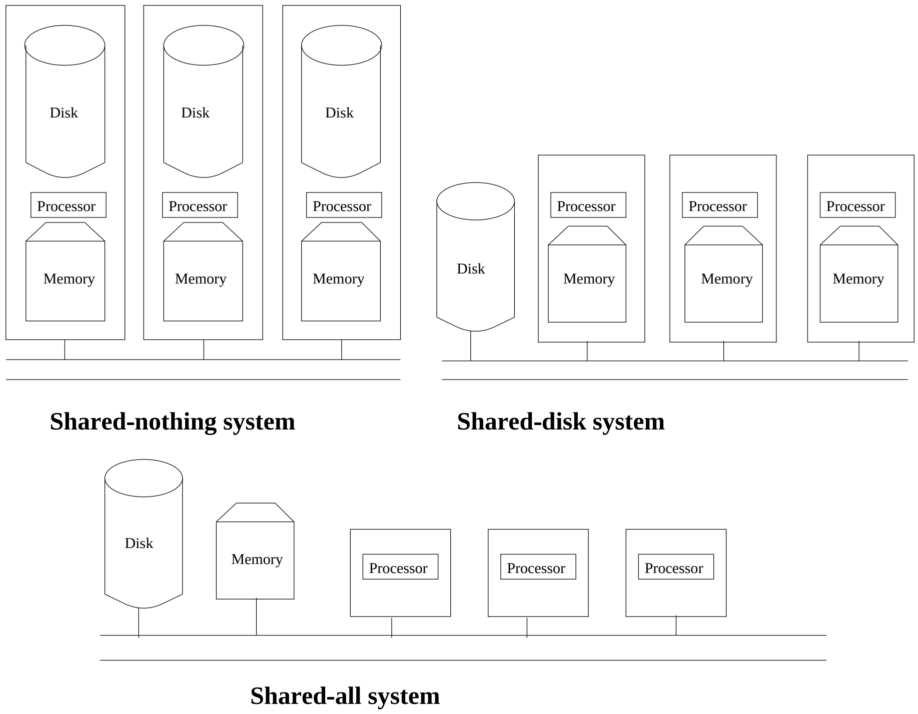


Figure 4: Different parallel architectures

so a natural decision for the logical data model is also object-oriented. Another possible solution is a relational database that has an object-oriented interface. The former is more flexible. The latter is easier to implement, but it can become limiting in the long run.

Building an object-oriented front-end to a relational database is a relatively simple task. The immediate advantage of the decision is that the theory behind relational databases is well known, and there are already implementations of fast relational databases. The disadvantage is flatness that relational database automatically creates to the system. In a system where objects are simple, it is possible to do simple mappings between objects and relations. The more complex the system becomes the harder the mapping becomes. Eventually the mapping becomes so complex and hard to maintain, that the system slows down to an unacceptable level. Currently the IN database mostly consists of simple structured data, although the amount of it can make some processing complex. However, we don't know what kind of objects are needed in the future. While we might get immediate advantage in choosing a front-end to a relational database, it could mean that later we would have to design and implement a totally new object-oriented database management system.

An ability to handle complex objects doesn't mean that the system is an object-oriented database. An object-oriented database system must satisfy two criteria: it should be a database management system and it should be object-oriented system. The authors in [Atki92] list 13 golden rules that any object oriented system should fulfill:

**Complex Objects** An object-oriented database must support objects that are built from the simple undivisible objects, such as strings and integers. Possible complex objects are tuples or structures, lists, and sets. A tuple is the simplest form, where a group of objects are joined under a common name. A tuple is also the basic building block of relational databases. A list and a set consist of a group of similar objects, that can be complex. A list is ordered, while a set is not.

**Object Identity** Object identity means that every object is an unique entity. If two objects have similar attributes in relational database they are considered a single object. In an object-oriented database they are two different objects. Object identity comes to the picture in object sharing. In complex objects, one part of the object can be an unique object alone. If it is shared between two objects, then in object model we know that the object is the same. In value-based models (such as relational model) we don't, because the values can be the same.

**Encapsulation** Encapsulation means that the specification and the implementation of an object are distinct. An object might show out different than it actually is. Some fields can be hidden, or a field can actually be a function that is counted every time it is referenced. When the object is stored to the database, both data and operations are stored.

**Types and Classes** A type in an object-oriented system tells what the abstract model of the object is like. It has two parts, the interface and the implementation. Together they define how the object looks like and how it behaves when referenced. Thus the type is a static definition of an object and its meta-information. The object itself can vary in the limits of its type.

A class is a similar term than type, but it has a more active role. A class contains two aspects: an object factory and an object warehouse. Class-based objects are created and maintained in runtime, and their role is free in most aspects. While the extra freedom gives more possible implementations, it also means that all compatibility checking must be done in real time.

**Class or Type Hierarchies** A class or type hierarchy means that an object can inherit some of its subobjects (fields) from another object type. Object inheritance is a very powerful modeling tool because it gives a chance to describe the world in more detail. It also helps implementations where the object inheritance is needed. In relational databases the inheritance model must be wired to the application programs. Such a solution is more accident prone and harder to maintain.

**Overriding, Overloading and Late Binding** Sometimes it is useful to have the same name to several different operations or fields. This becomes handy when it is combined with object hierarchies. For instance, let's suppose that there is a display function for all objects. According to the object type the function performs different operations. Some objects should be printed to a line printer, while others can be shown in interactive diagrams. In object-oriented system the high-level object type has the default display-function, while its subobjects have their own display functions. When referring to a subobject, its own display function is used. This is called overriding. It also means that the name display means several different functions that can be distinguished only from the object that owns the function. This is called overloading. Late binding means that the real operation used is decided at runtime, not at object creation time.

**Computational Completeness** Computational completeness means that the database management language (DML) can use any computable function. This is obvious in programming languages, where each language is as expressive. It is not obvious on database languages. For instance, SQL in relational databases is not computationally complete.

**Extensibility** Extensibility means that it is possible to define and create new types from old ones. There shouldn't be a distinction between old and new types, nor should a user notice when an old type is used instead of a new one.

The previous items described the necessary attributes that distinguish an object-oriented database from other logical model databases. The next ones distinguish an object-oriented database from an object-oriented programming language. As the IN database is a database, these items automatically belong to the definition.

**Persistence** Persistence means that data will stay between different executions. It is obvious in databases, but not in programming languages.

**Secondary Storage Management** One typical thing to databases is that the amount of data is high and it often grows during the database life span. The main memory available is not big and reliable enough to keep the data. Thus the database management system must be able to deal with secondary storage, such as disks. We will discuss more of this in chapter 6

**Concurrency** Concurrency means that the system should be able to let several simultaneous users to use the database without them interfering with each other. It is important in IN, where the response times must be kept low. Without concurrency one difficult query could block the whole system. We will discuss more of this in chapter 7.

**Recovery** The database should not loose any of the stored information in case of a failure. In IN database, the recovery level must exceed this; it is not allowed to let the users notice the failure at all. We will discuss more of this in chapter 7.

**Ad Hoc Query Facility** Ad Hoc Query Facility means that a user must be able to do simple interactive database queries. This is true to IN databases as well, although most queries will be done by a SCP and thus the language doesn't have to be that informative. We will discuss more of this in chapter 5.

# 5 Query languages

The IN database can be accessed either from a SCP or from a database maintenance system. In the former case the language can be in a low level, as long as it supports the queries and updates that are needed in the IN services, and it is extensible. It is important to ensure that query processing doesn't slow down the system. The fastest solution would be to do query optimization before the query is sent to the database.

The demand to query optimization before database access sounds strong, but actually it is not. A typical query from SCP to SDP is simple, and thus the query doesn't need much optimization. The updates from SCP to SDP are rare and simple. So in general the database access language doesn't need much resources to handle optimizing the queries and updates.

The database maintenance system is responsible of statistical queries and general database updates. The queries are made by the maintenance staff, and so a high level query language is needed. However, speed is not a critical issue in the language, because the maintenance systems are not usually time-critical.

Usually the chosen language to database management systems is SQL or some of its variations. In ODMG – 93 the language is called Object Query Language (OQL) [Catt94].

The authors in [Catt94] suggest the following principles and assumptions that also fit an IN high-level query language:

- OQL provides an easy access to the database. A simple interface is more important than computational completeness item OQL provides declarative access to the database
- OQL is object-oriented
- OQL has an abstract syntax
- the formal semantics of OQL can easily be defined
- OQL has SQL-like syntax, but it is possible to use other syntaxes by merging OQL with a high level programming language such as C++
- OQL provides high level access to sets of objects, and primitives to deal with structures and lists
- OQL does not provide explicit update operators. It relies on operations that are defined on objects. Thus every object class must have update operations, and
- OQL can be easily optimized.

We believe that the same principles work in an IN database query language. It is probable that the language is mostly used in embedded form, where it is part of a high level programming language. The interactive language interface itself is not among the most important subjects in the database design, because it is not used as often as in a regular database. Most queries are done through the SCP interface. The database maintenance staff needs powerful tools to access the database, and such tools can be built with the embedded query language.

## 6 Speed

Speed is the most critical aspect of the database. Although a good hardware configuration will lower response times, it is not enough to solve the real time response times.

The fastest way to reduce response times is to reduce disk access. Every disk access needs a magnitude more time than a memory access. Thus if data can be kept in main memory, it will lower the response times noticeable. Such a database is called a main memory database.

The idea of a main memory database has been introduced in [DeWi84] and [AmHK85]. Their idea is to move all data to the main memory and thus cut the disk access off completely. However, this ideal case is not realistic, as the amount of data usually exceeds the size of the main memory. Priorities are needed in order to decide what data is kept in main memory and what needs disk operations. Also a copy of the data in main memory must be kept in a disk to prevent data loss in case of a system failure. Usually an update is done both to the main memory and to some nonvolatile memory, where it can be moved to a backup disk with a lower priority process.

The physical location of the data doesn't necessarily affect the database management system architecture. In an ideal case only the stored data manager and the backup/recovery subsystem have to be changed to handle different data locations. However, optimizing the data locations while keeping it invisible to the system itself will make the subsystems more complex.

A main memory database should not use virtual memory, because the virtual memory algorithms are not optimized to heavy database access. The result can be disastrous to the speed. This lowers the amount of data that can be put to the main memory, as also the database software must be kept there. If virtual memory is used, totally new solutions are also needed. Such solutions have been introduced in literature, for instance in [ChSi92] and [KLVA93], but they need special virtual memory hardware that doesn't exist yet.

In IN databases the best solution is to keep the most time critical data in the main memory. It is not possible to keep all data in main memory, because history data alone exceeds the limits. One solution is to use priorities for data and queries. The lower priority data can reside on a disk, as long as it is reachable in few disk accesses. It is possible to use a lot of small parallel nodes to gain more main memory, but it is expensive. Thus the disks can't totally be replaced.

# 7 Transactions and recovery

A database communication is based on undivisible execution units called transactions. A transaction may have several database access operations that are either all accepted or none accepted. The database is consistent both before and after the transaction. Usually all resources that are needed to fulfill the read and write operations are locked before the transaction and released after it. If a resource can't be locked, the transaction is blocked until some other transaction releases the needed resource. This is called a two-phase commit protocol, and it was introduced in [EGLT76].

Transactions are needed for two reasons: They keep concurrent users from interfering with each other, and they keep the database consistent in case of a failure. A transaction manager, which is part of the concurrency control/recovery subsystem, is responsible of keeping the transactions in order. It also locks and releases resources.

The number of different operations in a single transaction, and the size of locked resources depend on the application in question. A rule of thumb is that transactions should be as short as possible, and yet keep logically undivisible units together. The size of a locked resource affects the concurrency level in a database. For instance, if every transaction locks the whole database then only one transaction can be active in a given time. If only an object is locked at a time then concurrent transactions can execute at the same time as long as they don't want to lock the same object. The drawback of this is extra overhead and extra data, as every object must have a lock attached to it.

There are at least three different types of transactions in an IN database:

1. Transactions that need only read main memory data, such as regular SCP-based transactions

2. Transactions that need to read disk data, or write main memory data, such as updating SCP-based transactions, SCP-based transactions that need extra data, and some statistical data gathering maintenance-based transactions.

3. Transactions that need to read several objects of disk data, or write to disk, such as transactions that backup main memory data to a disk, and most maintenance-based transactions.

The first class logical transactions are usually very short, and the extra overhead for handling object-level locks might be a major bottleneck in the system. Thus it is possible to let such transactions run alone in the database, as long as they are finished fast. This is true to almost all SCP-based transactions, especially when they use only data that resides in main memory. It's also easy to calculate the maximum time needed for such a transaction, which can help scheduling between parallel nodes. The second and third class transactions should use normal locking procedures, but in the second class the overhead should be minimized in cost of concurrency. Any SCP-based transaction should be finished as soon as possible. However, if all classes locked the whole database, the low-level transactions would block the database for unacceptable long times.

All write operations are saved into a transaction log during a transaction. Each of the operations consists of the old value and the new value. In case of a transaction failure, the transaction is canceled and all changes are returned back to the original values. A finished transaction also ensures that the changes it had made have been successfully stored to the database. In IN databases that keep the accessed data in main memory, a finished transaction can either mean that the data is written both to a disk and a main memory, or only to a main memory. In the former case, the data can be restored from the disks, unless a disk failure had happened, but it also means slower updates. In the latter case the updates are very fast, but the data can still be lost if a system failure occurs after a finished transaction.

Recovery problems are some of the most difficult to solve. The algorithms and solutions depend on the data model and the amount of main memory data used. The following failure situations can occur in a distributed IN database:

**Transaction failures** where a transaction can't be finished due to illegal operations in the transaction itself. The transaction is canceled and the database is returned to the original state. If the transaction would update several SDP-nodes, then it is better to write all updates to a secondary storage at first, and do the real updates to the database nodes after the transaction is validated. That way it is not necessary to send cancel commands over the network.

**Node failures** where a database node or part of it crashes. The simple solution is to block off the recovering node from the net and then reload all data. However, this might cause unwanted delay to the whole throughput. The other choice is to load data to the main memory when it is needed and index everything on the run. That way the whole recovery will take longer but it won't block the whole node. In any case, there has to be a backup system for avoiding data losses.

**Secondary storage failures** where a disk or other secondary storage system crashes. The data in the disk is lost, but in a fault tolerant system the system itself can function. If the database node can be shut down without affecting the general database throughput too much, then it is possible to restore the data from other database nodes and backup systems. This means that all data must be replicated to several nodes. If the node can't be shut down, then the replication must be made in the node itself. This can be done either in a subnode level in a parallel database, or in a disk level. In the first case, the other subnodes take care of the database activity while the disk is restored. In the second case, the other disks offer the same data, or the data can be restored from the other disks. This is often done in RAID-disks ([PaGK88],[Chen93]). A RAID-disk consists of several small disks that cooperate with each other. The data is stored the way that when any of the disks fails the data in it can be constructed from the others. The bottleneck of a RAID-system is its controller, which can also fail. The system performance can also lower drastically if one of the disks fails.

**Network failures** where a database node can't be reached due to a network failure. The recovery is left to the network maintenance system that is outside the database management system.

# 8 Summary

In this paper we have examined and analyzed what the current database research can offer to IN databases. In principle it is already possible to build a database system that fulfills the demands. A distributed database where each of the nodes is a parallel main memory database, and where the database languages and recovery systems are optimized to fast queries is a possible system. However, there are still a lot of open questions in areas such as data distribution and replication between nodes, preferable recovery algorithms, and query languages.

## References


[AiPW91] Aiken J. A., Parker S. T. and Woodwell D. R., Achieving Interoperability with Distributed Relational Databases. IEEE Network Magazine 5,1 (1991), pp. 38 – 45

[ApKS93] Appeldorn M., Kung R. and Saracco R., TMN + IN = TINA. IEEE Communications Magazine 31,3 (1993), pp. 78 – 85

[Atki92] Atkinson M., Bancilhon F., DeWitt D., Dittrich K., Maier D. and Zdonik S., The Object-Oriented Database System Manifesto. pp. 3 – 20 in Building an Object-Oriented Database System – the Story of $O_2$ (Bancilhon F., Delobel C. and Kannellakis P. eds). Morgan Kaufman Publishers, 1992

[Catt94] Cattel R. G. G., Object Database Standard: ODMG – 93. Morgan Kaufman Publishers, 1994

[CCIT92] Principles of Intelligent Network Architecture. Recommendation I.312 / Q.1201, CCITT, 1992

[Chen93] Chen P. M., Lee E. K., Gibson G. A., Katz R. H. and Patterson D. A., RAID: High-Performance, Reliable Secondary Storage. Technical report, University of California, Berkeley, 1993

[ChSi92] Chew K-M. and Silberschatz A., Toward Operating System Support for Recoverable-Persistent Main Memory Database Systems. Technical report, University of Texas at Austin, 1992

[EGLT76] Eswaran K. P., Gray J. N., Lorie R. A. and Traiger I. L., The Notions of Consistency and Predicate Locks in a Database System. Communications of the ACM 19,11 (1976), pp. 624 – 633

[ElNa89] Elmasri R. and Navathe S. B., Fundamentals of Database Systems. The Benjamin/Cummings Publishing Company, Inc., 1989

[Hade90] Haderle D. J., Database Role in Information Systems: The Evolution of Database Technology and its Impact on Enterprise Information Systems. pp. 1 – 14 in Database Systems of the 90s (Blaser A. ed). Springer-Verlag, 1990

[HoHa92] Homa J. and Harris S., Intelligent Network Requirements for Personal communications Services. IEEE Communications Magazine 30,2 (1992), pp. 70 – 81

[KLVA93] Krueger K., Loftesness D., Vahdat A. and Anderson T., Tools for the Development of Application-Specific Virtual Memory Management. ACM SIGPLAN Notices 28,10 (1993), pp. 48 – 64

[LoKM90] Lockemann P. C., Kemper A. and Moerkotte G., Future Database Technology: Driving Forces and Directions. pp. 15 – 34 in Database Systems of the 90s (Blaser A. ed). Springer-Verlag, 1990

[MoMa94] Molin K. and Martikainen O., Intelligent Network Tutorial for the Second Winterschool on Telecommunications. Technical report, Lappeenranta University of Technology & Telecom Finland, 1994

[PaGK88] Patterson D. A., Gibson G. and Katz R. H., A case for Redundant Arrays of Inexpensive Disks (RAID). pp. 109 – 116 in ACM SIGMOD Conference Proceedings. ACM, 1988

[Seli90] Selinger P. G., The Impact of Hardware on Database Systems. pp. 316 – 334 in Database Systems of the 90s (Blaser A. ed). Springer-Verlag, 1990